TP 5 – Création de processus

Vous devez faire au moins les exercices 1 à 7.
Un exemple de sujet d'examen concernant cette partie est donné dans les exercices 8 et 9.
L'exercice 10 est un approfondissement.

Exercice 1 - `fork`, `wait`

Le but de cet exercice est d'écrire le programme `hello`, dont le comportement est le suivant.

- Le programme principal (père) :
  1. crée un premier fils grâce à un appel à `fork`,
  2. crée un deuxième fils (toujours à l'aide de `fork`),
  3. écrit `"hello"` sur le flux standard de sortie,
  4. se termine.
- Le premier fils :
  1. attend 2 secondes,
  2. écrit `"\n"` sur le flux standard de sortie,
  3. se termine.
- Le deuxième fils :
  1. attend 1 seconde
  2. écrit `" world"` sur le flux standard de sortie,
  3. se termine.

1. Écrivez le programme `hello`, et testez-le.

2. Modifiez le programme `hello` pour que le père attende l'arrêt de ses deux fils avant de se terminer.

Exercice 2 - `fork`, `wait`, `sleep`

Le but de cet exercice est d'écrire le programme `sleepsort`, de sorte que

sleepsort n_1 n_2 ... n_k

écrive les entiers n_i (supposés positifs) dans l'ordre croissant sur le flux standard de sortie.

Pour cela, nous allons procéder de la façon suivante :

— Le programme principal (père) va créer un fils par entier n_i, puis attendre l'arrêt de tous ses fils.

— Le fils numéro i va attendre n_i secondes, puis écrire l'entier n_i sur le flux standard de sortie standard, et enfin se terminer.

1. Écrivez le fichier `sleepsort.c`.

2. Compilez et testez votre programme.

```
sh$  sleepsort 3 1 5 4 3
1
3
3
4
5
sh$
```

Exercice 3 - `fork`, `wait`, `pipe`

L'objectif de cet exercice est la réalisation d'une version parallélisée de la commande `wc -l`.

1. Écrivez le fichier `wc-par-v1.c`, selon l'approche suivante :
   - Le père calcule la taille `sz` du fichier f passé en argument (`argv[1]`), puis crée 4 fils et attend que ses fils s'arrêtent avant de se terminer.
   - Le premier fils ouvre le fichier f, lit les octets 0 (inclus) à $sz/4$ (exclu) de f, compte le nombre de sauts de ligne (caractère `'\n'`) de cette partie du fichier, et affiche le résultat sur le flux standard de sortie. Il traite ainsi la plage $[\![0,\ \frac{sz}{4}[\![$ du fichier f.
   - Le deuxième fils fait la même chose, mais sur la plage $[\![\frac{sz}{4},\ \frac{sz}{2}[\![$ du fichier.
   - Le troisième fils fait la même chose, mais sur la plage $[\![\frac{sz}{2},\ \frac{3\,sz}{4}[\![$ du fichier.
   - Le quatrième fils fait la même chose, mais sur la plage $[\![\frac{3\,sz}{4},\ sz[\![$ du fichier.

2. Testez votre implémentation et vérifiez que vos résultats sont cohérents avec ceux produits par la commande `wc -l`.

3. Faites un essai avec un fichier de 200 Mo placé dans `/tmp`.

```
sh$ dd if=/dev/urandom of=/tmp/bigfile.txt bs=1M count=200
sh$ wc -l /tmp/bigfile.txt ; wc-par /tmp/bigfile.txt
```

4. Ecrivez la variante `wc-par-v2.c`, dans laquelle les fils s'arrêtent avec comme code de retour le nombre de sauts de lignes lus. Le père pourra alors récupérer ces codes et en faire la somme.

   aide : Consultez le `man` de `wait` pour voir comment utiliser la macro `WEXITSTATUS`.

5. Quels sont les défauts de cette approche ?

6. Ecrivez la variante `wc-par-v3.c`, dans laquelle le père ouvre un pipe avant de créer les 4 fils. Les fils vont alors écrire leurs résultats dans ce pipe, de sorte que le père puisse les récupérer et en faire la somme.

7. Si vous avez du temps, écrivez la variante `wc-par.c`, dans laquelle le nombre fils passe de 4 à N.

   Faites varier N sur des exemples pour trouver la valeur optimale.

Exercice 4 - *broken pipes*, `dup2`

Cet exercice porte sur le programme `forkPN`, dont les sources sont disponibles dans `/pub/FISE_OSSE11/syscall/forkPN.c`.

1. Indiquez ce que fait le programme `forkPN`.

   Il crée deux ..... et deux processus .... qu'on appelera dans la suite `filsP` et `filsN`.
   Il lit des entiers sur le flux standard d'entrée :
   - Si l'entier lu est supérieur ou égal à 0, il l'envoie à son fils ..... via le pipe .....
   - si l'entier lu est inférieur ou égal à 0, il l'envoie à son fils ..... via le pipe .....
   - si l'entier lu est 0, il attend la ..... de ses .... avant de se terminer.

   Les fils, quant à eux, lisent des entiers sur leurs ..... respectifs les affichent et se terminent quand l'entier lu est 0.

   Le père affiche ses messages sur le flux standard ..... et les fils sur le flux standard .....

2. Compilez le code.
   ```
   sh$ gcc -Wall -Wextra -o forkPN /pub/FISE_OSSE11/syscall/forkPN.c
   ```

3. Lancez le programme comme indiqué ci-dessous pour avoir des entrées et des sorties lisibles :
   - Créez un nouveau terminal, récupérez le nom du *tty* de ce terminal, puis faites en sorte que le terminal n'affiche plus rien et qu'il puisse être fermé à l'aide <CTRL-D>.
     ```
     sh$ xterm -e sh -c 'tty ; cat >/dev/null' &
     ```
   - Dans votre terminal initial, lancez `forkPN` en redirigeant le flux standard de sortie sur le *tty* du nouveau terminal
     ```
     sh$ ./forkPN >/dev/pts/N
     ```
     où N est écrit en haut du second terminal.

4. Écrivez la variante `forkPN-v1.c`, similaire à `forkPN.c`, mais où le père comme les fils s'arrêtent lorsqu'ils ont atteint la fin de leur fichier de lecture.

   Le père attend toujours la fin des fils pour se terminer.

   aide : Il faut que les fils détectent la fin de fichier lors de la lecture sur leur pipe. Cela ne peut arriver que lorsqu'il n'y a plus d'écrivains pour le pipe en question.

5. Écrivez la variante `forkPN-v2.c`, similaire à `forkPN-v1.c`, mais où le père réagit à l'arrêt d'un de ses fils en tuant l'autre fils et en écrivant sur le flux standard d'erreur un message du genre :

   `"pere:pid: fin inattendue du fils <pid>`
   `pere:pid: arrêt du fils <pid>`
   `pere:pid: terminaison".`

aide :

— L'arrêt d'un des fils peut être provoqué manuellement par la commande :

```
sh$ kill -TERM <pid>
```

où `<pid>` est le PID du fils visé.

— Si chaque fils est le seul lecteur pour le pipe correspondant, le père peut détecter l'arrêt d'un fils lors de l'écriture dans le pipe.

Dans ce cas, si le fils se termine, il n'y a plus aucun lecteur sur le pipe et une écriture dans ce pipe provoque une erreur d'écriture et l'envoi d'un signal `SIGPIPE`.

Solution 1 : Définir un gestionnaire pour le signal `SIGPIPE` qui va se charger d'arrêter les deux fils. Le code de retour de `kill` permettra de savoir si le fils visé était déjà arrêté.

Solution 2 : Tester si l'écriture dans le pipe provoque une erreur. Dans ce cas, on sait que ce pipe n'a plus de lecteur, donc que le fils correspondant s'est arrêté. On peut alors arrêter l'autre fils.

6. Écrivez une variante `forkPN-v3.c` du **fichier d'origine** `forkPN.c`, dans laquelle :
   — Le père arrête de lire quand il reçoit le signal `SIGINT`.
   — Le père envoie alors le signal `SIGUSR1` à ses deux fils.
   — Sur la réception du signal `SIGUSR1`, les fils se terminent en écrivant un message du type
   `"fils:pid: fin due à la réception de SIGUSR1".`
   — Le père attend la mort de ses fils et écrit le message
   `"pere:pid: terminaison".`

aide : Vous pouvez envoyer un signal `SIGINT` en tapant <CTRL-C> dans le terminal. Dans ce cas, le père et les fils reçoivent ce signal. Le comportement par défaut étant d'arrêter les processus, il faudra penser à changer de gestionnaire pour `SIGINT` aussi au niveau des deux fils.

Exercice 5 - processus, threads, `time`

Cet exercice porte sur les programmes `runcmd-fork` et `runcmd-thread`, dont les sources sont disponibles dans `/pub/FISE_OSSE11/syscall/runcmd-fork.c` et `/pub/FISE_OSSE11/syscall/runcmd-thread.c`.

Ils sont fonctionnellement semblables :

1. Ils allouent n mégaoctets de mémoire (n étant la valeur du 1er argument passé au programme) ;
2. Ils répètent 1000 fois la suite d'actions suivante :
   — création d'un fils qui affiche `hello` sur la sortie standard
   — attente de la fin du fils.

Comme les noms le suggèrent, ces deux programmes diffèrent par leur implémentation :

— runcmd-fork crée ses fils (processus lourds) avec `fork`,
— runcmd-thread crée ses fils (processus légers) avec `pthread_create`.

1. Compilez les deux programmes.

```
sh$ gcc -Wall -o runcmd-fork /pub/FISE_OSSE11/syscall/runcmd-fork.c
sh$ gcc -Wall -pthread -o runcmd-thread /pub/FISE_OSSE11/syscall/runcmd-thread.c
```

2. Mesurez les temps d'exécution des deux programmes avec une mémoire allouée de 1 Mo.

```
sh$ time ./runcmd-fork 1 >/dev/null
sh$ time ./runcmd-thread 1 >/dev/null
```

   On redirige ici le flux standard de sortie dans `/dev/null` afin de ne pas mesurer le temps lié à l'affichage (qui n'est pas du tout négligeable, et qui n'apporte rien pour la suite).

3. Recommencez ces mesures pour une mémoire allouée de 10, 100, 200, 500 et 1000 Mo.

4. Expliquez les mesures obtenues.

Exercice 6 - pthreads

Le but de cet exercice est de reprendre ce qui a été fait dans l'exercice 3, et de proposer une nouvelle variante reposant cette fois sur des *pthreads* (à la place des processus lourds créés par `fork`).

1. Ecrivez la variante `wc-par-v4.c`, dans laquelle le père crée les 4 threads. Chaque thread va compter et renvoyer le nombre de sauts de lignes dans la partie du fichier qu'il doit traiter.

   Le père attend la terminaison des threads, récupère les différentes valeurs renvoyées, et affiche leur somme sur le flux de sortie standard.

   aide :

   Le troisième argument attendu par `pthread_create` est une fonction de prototype

   `void* start_routine(void*)`

   c'est-à-dire une fonction qui :

   — prend en entrée un pointeur vers une zone mémoire contenant les arguments,

   — renvoie un pointeur vers la zone mémoire où le résultat a été stocké.

   Le type `void*` est utilisé ici comme *joker*, puisqu'il n'est pas possible de savoir à l'avance quels vont être les besoins en terme de types pour la routine choisie par l'utilisateur.

   Pour gérer les arguments des différents threads, une approche simple est de définir une structure (type record) genre

   ```
   struct arg {
       char* file;
       int begin;
       int end;
   };
   ```

   On peut alors lancer les quatre threads de la façon suivante :

   ```
   pthread_t thids[4];
   struct arg args[4];

   for (int i = 0; i < 4; ++i) {
       args[i].file  = argv[1];
       args[i].begin = ... ;
       args[i].end   = ... ;
       pthread_create(&thids[i], NULL, start_routine, &args[i]);
   }
   ```

   Enfin, pour la valeur de retour, on veillera à allouer la mémoire nécessaire via un appel à `malloc` dans le code de `start_routine`. Une fois que le père aura récupéré cette valeur de retour, il devra libérer la mémoire via un appel à `free`.

2. Ecrivez la variante `wc-par-v5.c`, dans laquelle le père crée les 4 threads. Chaque thread va :

   — compter le nombre de sauts de lignes dans la partie du fichier qu'il doit traiter,

   — mettre à jour un compteur global.

   Le père attendra la terminaison des threads qu'il a lancé, puis affichera la valeur finale du compteur.

   On utilisera un sémaphore (type `pthread_mutex_t`) pour s'assurer que le compteur global n'est modifié que par un seul thread à la fois.

Exercice 7 - pthreads, mutex

Le programme `hello` a pour but d'écrire le message `"hello"` sur le flux standard de sortie. Pour cela, il crée deux threads :

- Le premier thread écrit `'h'` et `'o'`.
- Le deuxième thread écrit `'ell'`.
- Le programme principal écrit le `'\n'` final.

1. Écrivez le code `hello_sleep.c`, dans lequel la synchronisation des écritures est réalisée grâce à des appels à `sleep`.

2. Écrivez le code `hello_mutex.c`, dans lequel la synchronisation des écritures est réalisée grâce à des sémaphores (type `pthread_mutex_t`).

Exercice 8 - `fork`, `pipe` et flux noyaux **(examen 2017)**

Considérons le programme ci-dessous.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


void fils(int com[], int fin) {
    int ret;
    char c;

    while ((ret = read(fin, &c, 1)) == 1) {
        if (write(com[1], &c, 1) == -1) { perror("[fils.write]"); exit(1); }
    }

    if (ret == -1) { perror("[fils.read]"); exit(2); }
    exit(0);
}

int main() {
    int ret;
    char c;
    int com[2];

    if (pipe(com) == -1) { perror("[pipe]"); exit(1); }

    int in  = open("in",  O_RDONLY);
    int out = open("out", O_WRONLY | O_TRUNC);
    if (in == -1 || out == -1) { perror("[open]"); exit(2); }

    if ((ret = fork()) == -1) { perror("[fork]"); exit(3); }
    if (ret == 0) fils(com, in);

    if ((ret = fork()) == -1) { perror("[fork]"); exit(4); }
    if (ret == 0) fils(com, in);

    while ((ret = read(com[0], &c, 1)) == 1) {
        if (write(out, &c, 1) == -1) { perror("[write]"); exit(5); }
    }

    return 0;
}
```

1. Indiquez ce que contiendra le fichier `out` après l'exécution du programme.

2. Ce programme ne s'arrête pas, indiquez pourquoi.

3. Proposez une correction pour qu'il s'arrête.

Exercice 9 - père, fils et petit-fils **(examen 2017)**

Le programme `greeting` a pour but d'écrire le message "comment ca va ?" sur le flux standard de sortie. Pour cela :

— Le programme principal (*père*) crée un processus fils ;

— Ce *fils* crée lui aussi un processus fils, qu'on appelera *petit-fils*.

Concernant l'affichage :

— Le petit-fils se chargera de la partie `"comment"` ;

— Le fils se chargera de la partie `" ca "` ;

— Le père se chargera de la partie `"va ?"`.

1. Écrivez le code de ce programme, en faisant en sorte qu'il affiche bien `"comment ca va ?"` quelque soit l'ordonnancement des différents processus.

   Le programme suivra les usages standards d'Unix et n'utilisera que des **flux noyau**.

Exercice 10 - Mini shell

L'objectif de cet exercice est la réalisation d'un mini shell.

On va se concentrer ici sur les problématiques de programmation système soulevées par la conception d'un shell. En revanche, on ne traitera pas de la gestion précise de la syntaxe shell (de fait, pour faire les choses correctement, il faudrait utiliser les outils présentés dans l'UE `ASCO23` au semestre 3).

En fait, on va même s'autoriser à utiliser une syntaxe un peu différente et très simplifiée :

- On ne lancera que des commandes sans arguments ;
- On ne gérera pas les variables, sauf quelques cas très particuliers ;
- Les symboles `<`, `>`, `|` et `&` seront placés au début des commandes, de sorte que le traitement à effectuer puisse être déterminé en fonction du premier caractère saisi par l'utilisateur ;
- Les redirections seront mises en place en amont des commandes, sur des lignes dédiées.

1. Écrivez le code `minishell.c` d'un programme qui lit des commandes sur le flux standard d'entrée et les exécute en avant plan :
   - Le programme attend la fin de la commande courante avant de lire et exécuter la suivante ;
   - Les commandes n'ont pas d'arguments ;
   - Le programme écrit un message d'erreur explicite si la commande saisie n'a pas été trouvée ;
   - Le programme s'arrête sur la fin du fichier standard d'entrée ou si la commande **exit** est saisie.

   On pourra faire des tests avec des commandes comme `ls`, `cat`, `yes` ou encore `./a.out` après avoir compilé un fichier source en `C` de votre choix.

2. Complétez le programme précédent pour gérer les commandes suivantes :

   `$$` qui affiche le PID du programme,

   `$?` qui affiche le code de retour de la dernière commande exécutée.

3. Complétez le programme pour gérer les *redirections* :

   `>file` redirigera le flux standard de sortie de la prochaine commande dans `file`,

   `<file` redirigera le flux standard d'entrée de la prochaine commande afin de lire dans `file`.

   On veillera à ce que la redirection soit appliquée **uniquement** à la prochaine commande. Et on ne gérera que le cas où il n'y a pas d'espace entre le symbole de redirection et le nom du fichier.

4. Complétez le programme pour introduire la *mise en arrière-plan* :

   `&cmd` lancera la commande `cmd` en arrière plan.

   Le programme lira alors la commande suivante sans attendre.

5. Complétez le programme pour que la commande `fg` fasse passer en avant plan la dernière commande lancée en arrière plan.

   note : Dans un premier temps, on pourra se contenter de gérer uniquement le cas d'un unique processus en arrière plan, et renvoyer un message d'erreur si l'utilisateur demande de lancer une seconde commande en arrière plan.

6. Complétez le programme pour que la suite de commandes `|cmd1` puis `cmd2` lance `cmd1` et `cmd2`, avec le flux de sortie standard de `cmd1` utilisé comme flux d'entrée standard pour `cmd2`.

   Il s'agit donc d'avoir un comportement équivalent à

   sh$ cmd1 | cmd2

   dans un shell standard.

   On pourra tester avec `|top`, suivi de `head` / `tail` / `less`.

7. Complétez le programme pour que la saisie de <CTRL-C>, qui génère le signal `SIGINT`, ne termine ni `minishell`, ni les processus en arrière plan, mais uniquement le processus en avant plan si il existe.

> **aide :** Pour cela, il faut que
>
> 1. le processus `minishell` ignore le signal `SIGINT`,
> 2. le processus en avant plan traite le signal `SIGINT` avec le comportement par défaut,
> 3. le signal `SIGINT` ne soit pas transmis aux processus en arrière plan.
>
> Une manière de réaliser le dernier point est de faire appel à `setpgid(0, 0);` ce qui a pour effet de placer le processus courant dans un nouveau groupe de processus.

8. Complétez le programme précédent pour que la saisie de <CTRL-Z> suspende le processus en avant plan uniquement, et pas le `minishell` ni les processus en arrière plan.

> **aide :**
>
> 1. Le noyau envoie le signal `SIGSTP` à tous les processus du groupe de processus ciblé.
> 2. Le processus `minishell` doit attraper ce signal.
> 3. Le processus en avant plan doit avoir le traitement par défaut (suspendre) pour ce signal et appartenir au groupe qui contrôle le tty. Un appel à `fork` met automatiquement le fils dans le groupe du père.
> 4. Les processus en arrière plan ne doivent plus appartenir au groupe (ceci a normalement déjà été mis en place à la question précédente). Ils ne reçoivent alors pas le signal `SIGTSTP`.
> 5. La commande `fg` fait repasser en avant plan soit le processus en avant plan précédemment suspendu, soit le dernier processus lancé en arrière plan. Dans le deuxième cas, il faut remettre le processus dans le groupe pour que <CTRL-Z> soit de nouveau actif. Pour ce faire, on utilisera les fonctions `getpgid` et `tcsetpgrp`.